Swarm Rule-base Development using Genetic Programming Techniques

Michael A. Kovacina[1,3], Daniel W. Palmer[2], Ravi Vaidyanathan[1,3]
[1]Orbital Research Inc. Cleveland, OH, USA
[2]John Carroll University, University Heights, OH, USA
[3]Case Western Reserve University, Cleveland, OH, USA[1]

Abstract— The programming of a swarm of autonomous agents to perform a given task is a time-consuming process. This work proposes the development of a genetic programming system to self-evolve rule-bases for swarms of autonomous agents. The genetic programming system utilizes finite state machines to represent candidate algorithms. The currently implemented system has been used to evolve rule-bases for the problem of object collection, a standard problem in the area group coordination.

***Index Terms*—genetic programming, swarm**

I. Introduction

Developing a high-level rule-base for the completion of a task by a single autonomous agent can be daunting. Trying to program a large group, or swarm, of less capable agents to perform the same task can be even harder. The programming of a swarm of autonomous agents to perform a given task is a time-consuming process that poses many hidden (or emergent) properties that must either be exploited or repressed. This work explores the use of genetic programming to evolve rule-bases for swarms of autonomous agents.

II. Genetic Programming Implementation

Genetic programming [1] is an extension of traditional genetic algorithms. Typically, a population in a genetic algorithm consists of encoded solution strings. Genetic programming populations, however, consist of encoded programs, which when executed provide solutions to the problem being attacked. Fundamentally, the principal difference between the two is that genetic algorithms tend to provide one specific solution, whereas genetic programming provides an algorithm to find a solution.

Traditionally, genetic programming methods can be broken down into four major components: solution encoding, evolutionary operators, fitness evaluation, and selection mechanisms. This next section will discuss each of these components and how they have been implemented for use in evolving rule-sets for swarms of agents.

A. Solution Encoding

The encoding of candidate solutions is perhaps the most important parameter to be determined when implementing an evolutionary system. In the case of swarm encoding, however, a definition for an agent must be established prior to the characterization of any scheme. For the purpose of this work, an **agent** is defined as an entity that perceives its environment through *sensors* and affects its environment through *actions*. All the sensors used are binary sensors, and all of the actions implemented are assumed to be atomic. The agents also have a small amount of memory to act as persistent storage for certain actions, but this memory is not currently used to influence state changes. This definition of an agent is very abstract on purpose since the focus of this work is to evolve rule-sets for higher-level behaviors. Consequently it is critical to abstract away the lower-level details.

A finite state machine representation was chosen for the solution-encoding scheme. A finite state machine consists of a collection of states, with a set of transition rules associated with each state. Each of these transition rules is then associated with a particular action and the name of the next state in which the machine should reside. Thus, a finite state machine can be seen as a collection of subroutines that call other subroutines based on given inputs. The finite state machine representation was chosen for its simplicity and clear representation of agent states. The internal representation of a finite state machine can be seen as a 4-tuple consisting of the set: {*current_state*, *sensor_states*, *action*, *next_state*}. The variables *current_state* and *next_state* are simply names of states within the finite state machine, and signify the current agent state and the state that it should transition to if the rule fires. The *action* is the name of the action that will be executed if the transition rule is fired. The *sensor_states* variable is represented as a binary string, with a '1' signifying that a sensor response is true and a '0' signifying that a sensor response is false.

Fig. 1 depicts an example finite state machine that will direct an agent to pick-up an object and bring it to a light source. The finite state machine has two states, S1 and S2. The agent is endowed with two sensors, one for detecting nearby light sources and another for detecting objects within picking up range. There are three actions that may be performed by the agent: move randomly, pick up and object, and put down an object. The transitions between states (shown by arrows), define the conditions on which an agent changes

[1] Correspandce: rxv@cwru.edu

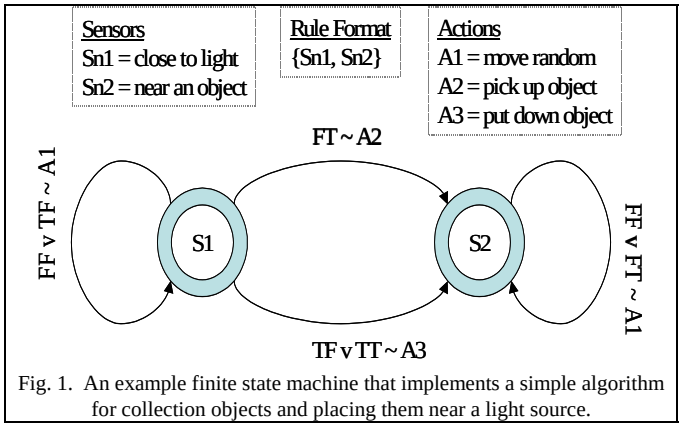

Fig. 1. An example finite state machine that implements a simple algorithm for collection objects and placing them near a light source.

state. A condition is denoted though a binary representation which translates into the desired sensor conditions [2]. For example, the condition denoted by the rule "FF v TF", says in words, "IF (not near a light source and not near an object) OR (near a light source and not near an object) THEN (move randomly)."

With this finite state machine, an agent will begin execution in state S1. While an agent does not see an object, the agent will randomly walk and stay in state S1. When an object comes within reach, the agent will pick up the object (action A2), and transition into state S2. Once in state S2, the agent will walk randomly as long as there is not any nearby light sources. Once a light source is found, the agent will put down the object and return to state S1, beginning the process over again.

B. Evolutionary Operators

1) Mutation

In this work, the designed mutations affect small changes upon single solutions within a given population, thus "fine-tuning" a solution. The creation and implementation of a mutation is tightly coupled with the solution encoding. Since the solutions will be encoded as finite state machines, the mutations will effect the number and composition of the states in the finite state machine. **Table I** contains a listing of the various mutations used in this work.

It should be noted that some of the implemented mutations modify the fundamental structure of the finite state machine. Thus, if left unchecked, the finite state machines would soon degenerate and have little functionality. Measures were therefore taken to correct any destructive changes to the mutated finite state machine. For example, when a 'remove state' mutation is applied to a solution (e.g. remove state s_1), any transition whose *next_state* is s_1 will be assigned a random state from the machine as the new *next_state*.

2) Crossover

The crossover operation first selects two solutions from the population to be parents, and then creates a new solution by combining elements of the two selected parents. Within the finite state machine representation, there are two options to implement crossover operations; transition-based and state-based. In the transition-based method, the parents are interpreted as a list of transitions, and thus the crossover point can occur in the middle of a state. With the state-based method, the parents are seen as a sequence of states, so when crossover occurs entire states are preserved.

Both methods for the crossover operation were used and their performance contrasted. The transition-based method contributed more when there were fewer mutations due to its ability to modify states by selecting the crossover point within a state. The state-based method was more effective when a greater number of mutations were present since preserving entire states allows states to evolve over time without being catastrophic modification.

Fig. 2 shows a typical crossover operation within the state-based representation. In this case, the states and transitions of the finite state machine on the right are preserved through crossover, with only the additional state (3) and its transition to state (1) added to the child shown. The preservation of actions along with states results in safeguarding structures within the FSA while also enabling significant changes as well (*i.e.* entire states and actions are added or subtracted from parents in the following generation).

The transition rule-base representation was designed to adjust actions based on sensor feedback (transitions) between states. **Fig. 3** shows the same parents in the previous figure performing a typical crossover operation under the transition-based representation. In this crossover, states from both parents are preserved, yet a variable number of transitions between the two survive in the child. This structure allows for finer manipulations of actions within the FSA through genetic union.

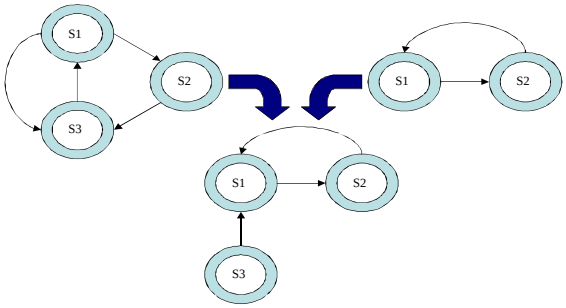

Fig. 2. An example of state-based crossover for two finite state machines.

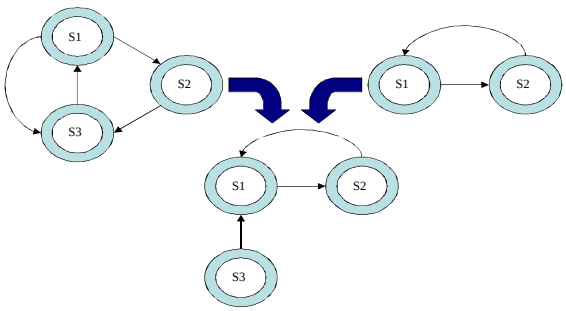

Fig. 3. An example of transition-based crossover for two finite state machines.

Given the broad level of swarm behaviors that need to be captured, these two structures are designed to complement one-another based on the evolutionary progress of the population. For example, should a partially effective solution be reached based on the broader changes allowed by the state-based representation, an evolutionary plateau would also likely occur given the lack of flexibility necessary to modify transitions. In such a case, switching to a transition-based representation would likely spur progress towards a final solution.

C. Fitness Evaluation

For fitness evaluation, a fitness function is defined that assigns a score to each candidate solution based on their ability to satisfy certain criteria. A fitness function should highlight the salient features of a problem in order to effectively drive evolution. The fitness functions used here were actually a combination of many fitness functions, one for each salient feature. Then each score was weighted and the sum of the weighted scores was assigned as the final score for the evaluated solution.

In order to evaluate a solution, the rule-base must be executed. SWEEP [3] (*SW*arm *E*valuation and *E*xperimentation *P*latform) was used to execute each rule-base. The finite state machine was given to SWEEP, which then created a swarm of agents, each with the given finite state machine as an internal controller. Each run of SWEEP provided a sequence of time-step snapshots, and it was these snapshots that were used in generating the final fitness scores.

There were two methods used to obtain the data needed for fitness evaluation. The first method sampled every n^{th} time step from the solution output, and combined the scores from all n samples as the final score for the solution. This method is better when one is trying to evolve behaviors that depend on the movement and position of the agents, such as pattern formation. Unfortunately, the sampling method can slow down the system significantly if the sampling rate is too high. The second method only looked at the state of the solution in the final time step. This method was much faster, and excelled in evaluating solutions for problems that were independent of the final position of the agents and focused more on the state of the environment, such as object collection.

TABLE I
FINITE STATE MACHINE MUTATION OPERATORS

Mutations
Add a state to the finite state machine
Remove a state from the finite state machine
Add a transition to a state
Remove a transition from a state
Modify the *next_state* associated with a transition
Modify the expected sensor values for a transition
Modify the action associated with a transition

D. Selection Mechanisms

Selection mechanisms are used to filter out poor solutions and too keep a population at a desirable size. Two methods were used during this work. The first selection mechanism is the traditional fitness proportional or "roulette" method. In this method, the probability that a solution will survive into the next generation is proportional to their fitness score. Thus, more fit individuals are more likely to survive, but some less fit individuals also survive. This is an important aspect because sometimes a solution is unfit overall, but a portion of that solution when combined with a more fit solution can produce offspring that is more fit than either of the parents. The second selection method used was elitism, which effectively selects only the best solutions for survival.

Neither method clearly made evident that it was the best selection mechanism, with both providing about the same performance in terms of convergence rate. In the end, a hybrid method was used that allowed the top few solutions to survive, and then used roulette selection to select the rest of the solutions for survival. A more in-depth trade-off study of different selection mechanisms will be performed in conjunction with this work at a later date.

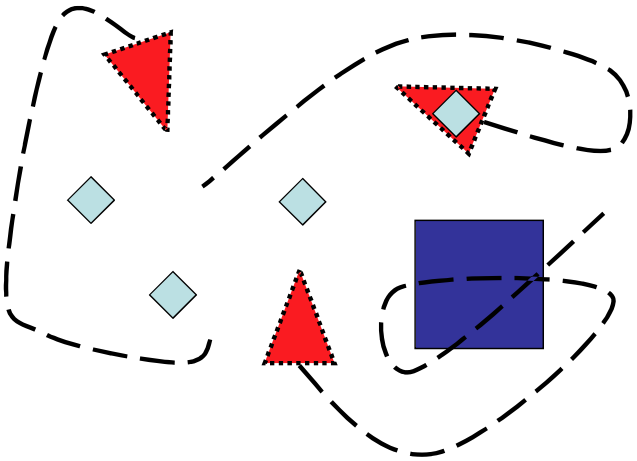

Fig. 4. Three agents are attempting to collect four objects scattered over the environment. Once an agent has picked-up an object, the agent finds the goal area and delivers the object.

III. CASE STUDY: OBJECT COLLECTION

In order to validate the functionality of the system, the standard group coordination problem of object collection was considered. Object collection is a very prevalent problem in the world, with examples ranging from the foraging techniques of ants to the optimized collection routes for garbage men.

A. Problem Description

Given *N* agents and *M* objects randomly distributed over a grid-based environment with pre-defined square goal region, collect and place all of the objects inside of the goal within a given amount of time *T*. An example scenario is shown in **Fig. 4**.

There were several reasons for selecting object collection as

a driving problem. First, Koza [1] selected object as a problem for evolving emergent behaviors, and so it was hoped that at some point a comparison could be examined between Koza's results and the results of this work. Secondly, the fitness of a solution can be adequately determined solely through the examination of the last time-step, thus reducing the amount of computation time needed. Finally, the salient features of the fitness function are easily identifiable and do not conflict with each other, which allows for the easy creation of fitness functions for this problem.

TABLE II
AGENT ACTIONS AND SENSORS

Actions	
Name	Description
A_1	Move pseudo-randomly
A_2	Move { up, down, left, right }
A_3	Move towards the nearest object
A_4	Move towards the goal
A_5	Pick-up an object
A_6	Put down an object, if holding one

Sensors	
Name	**Description**
S_1	Near an object
S_2	On top of an object
S_3	Near the goal
S_4	In the goal

B. Implementation

The system was set to continue evolving until a "perfect" solution was evolved. The population size was kept at 25. Normally, population sizes are much larger, but due to the simulation component of the evaluation, smaller populations were used to speed-up overall execution [4]. The mutations and crossover operations previously defined were used, with a mutation rate set at 50% and a crossover rate of 50%. When mutating, each possible mutation had an equally probable chance of being selected.

The environment for object collection was defined to be a 100x100 grid, with the goal region being a randomly placed 5x5 square. There were 30 agents and 30 objects randomly distributed over the environment. So as not to create any early false positives, it was assumed that no objects would have an initial position inside of the goal region.

Specific sensors and actions were created for the object collection. Four sensors and six actions were defined for an agent performing object collection, as listed in **Table II**.

Both sensors S_1 and S_2 have effective radii of 3 grid units. By separating the sensor logic into "near" something and "on/in" something, the rules for state transitions can be kept in the binary realm.

What is meant by the phrase "pseudo-random" in action A_1 is that a random heading is chosen, and that heading is then followed for a randomly chosen number of time steps. The heading and current step count is stored within agent memory so that the information can persist between time steps. Actions A_3 and A_4 convenience functions because the focus of the system is to evolve high-level logic, not low-level controls.

C. Preliminary Results

The results of this work are best understood through the development process of the fitness functions. The initial fitness functions focused on the final goal, that being collecting all objects into the goal. While this function did convey the intent of the problem, it did not sufficiently describe all the salient features of the problem and thus did not drive evolution quickly enough. Later fitness functions more precisely defined the problem at hand and thus were able to more efficiently drive the evolutionary process. The following sections will discuss the three main fitness functions that were used in this work.

1) Fitness Function #1

The first fitness function looked solely at the number of objects that were within the goal region at the final time step. Each solution was given one point for every object found in the goal region. Due to the nature of this fitness function, a solution can only receive points once it has evolved a partially correct solution, meaning that at least one agent picked-up an object and dropped it in the goal. This unfortunately does not imply that the agent dropped the object in the goal because it recognized the goal region as desirable, as evidenced by the false positive solutions found in which an agent picked-up an object close to the goal and just happened to randomly drop the object in the goal.

Overall, this fitness function performed poorly, producing hundreds of generations with an average score of zero. Once serendipity struck though, the fitness function quickly singled out well-performing solutions. Unfortunately, serendipity is not an event that one should wait for when fast performance is desired. Thus, focus was returned to the statement of the problem, which in turn lead to the development of a more precise fitness function as discussed in the next section.

2) Fitness Function #2

The development of the second fitness function was a result of a closer examination of the problem statement. This fitness function assigned 31 points for every object in the goal and 1 point for every object being held by an agent during the final time step. The number of points assigned for each object in the goal was computed as '1+(number of objects)' because this way, a solution which places one object in the goal region has a higher fitness score than a solution that picks up every object but fails to place any objects in the goal region.

This fitness function performed much better than the previous fitness function, being able to produce solutions that successfully collected every object within 50 generations. Unfortunately, evolution stalled as the population of solutions converged on solutions that successfully collected all the objects, but did not place any in the goal. This once again can be seen as a deficiency in the fitness functions ability to clearly convey the full intent of the problem to the evolutionary process. In this case, the fitness function did not provide a clear evolutionary path between the stages of picking up an object and placing the object into the goal. Thus, focus once again turned to a closer examination of the problem statement to produce a better fitness function.

3) Fitness Function #3

The final fitness function developed identified the three

main salient features of object collection that can be evaluated looking only at the final time step. These three features are: pick up objects, bring objects to the goal, and put down objects in the goal. As in the previous fitness functions, a single point was given to a solution for every object that was being held by an agent during the final time step, and 31 points were given for every object put down in the goal. As for bringing the objects to the goal, 15 points were awarded to a solution for each agent holding an object in the goal during the final time step. Since this aspect is the intermediary stage, it followed that the amount of points awarded should be approximately in the middle.

This fitness function performed the best overall. Much like the previous fitness function, solutions that collected all objects were quickly evolved. Instead of stagnating at this stage, as in the previous fitness function, evolution continued and solutions emerged that not only collected all the objects in the environment, but also saw agents holding objects in the goal area. So, this fitness function was successful in collecting and clustering the objects, but in the end, it was once again left up to serendipity to make the final evolutionary jump (most likely though mutation) from an agent just holding an object in the goal to the agent actually putting down the object in the goal.

IV. Future Work

Three main avenues remain to be explored before problems harder than object collection can be tackled. First, even more precise fitness functions can be constructed by switching from a final time step method to a sampling method. For example, another component to a fitness function can be added such that solutions receive rewards for agents that move in the direction of the goal. The second avenue of exploration is that of adapting mutation and crossover rates. Since mutations affect fine-grained changes and crossover affects coarse-grained changes, these two aspects can be exploited during different stages of evolution. For example, during a period of stagnation, the percentage of crossover could be increased in order to produce more diverse solutions to break free of the evolutionary halt. The final avenue to explore is that of incorporating a-priori knowledge into the system. In many cases, the user will be privy to domain-specific knowledge. For example, in object collection, the user knows that once an agent carries an object into the goal region, the object should be placed down. This piece of information could be inserted into a mutation that inserts that piece of knowledge into the solution. Over time, this could be eventually be implemented in "smart" mutations that can recognize which solutions need this information, and possibly even determine where in that solution this information should be inserted.

V. Conclusion

The purpose of this work was to develop a genetic programming system to evolve rule-sets for groups or swarms of agents. Solutions were encoded as finite state machines, and the evolutionary operators (mutations and crossover) were implemented to operate on finite state machines. The standard group coordination problem of object collection was explored as a driving problem for the development of the system.

Through the development of the object collection scenario, the need for a fitness function to clearly communicate the salient features of the problem to the evolutionary process became evident. The final fitness function used clearly delineated the three stages of object collection, provided a weighted score for each solution based on these stages, and successfully evolved a rule-set for object collection.

Acknowledgements

The authors gratefully acknowledge the generous support of the Air Vehicles Research Directorate at Wright-Patterson Air Force Base under the direction of Mr. Daniel Schreiter and Mr. Bruce Clough.

References

[1] J. Koza. *Genetic Programming*. Cambridge, MA: MIT Press, 1992.
[2] J. Holland, *Hidden Order*. Cambridge, MA: Perseus Books, 1995.
[3] M. Kovacina, D. Palmer, G. Yang, R. Vaidyanathan, "Multi-agent Control Algorithms for Chemical Cloud Detection and Mapping Using Unmanned Air Vehicles", In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE*, v. X, 2002.
[4] C. Frey, G. Leugering. "Evolving Strategies for Global Optimization – A Finite State Machine Approach," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, pp. 27-33, 2001.*